Heracles: Improving Resource Efficiency at Scale

David Lo[†], Liqun Cheng[‡], Rama Govindaraju[‡], Parthasarathy Ranganathan[‡] and Christos Kozyrakis[†]
Stanford University[†] Google, Inc.[‡]

Abstract

User-facing, latency-sensitive services, such as websearch, underutilize their computing resources during daily periods of low traffic. Reusing those resources for other tasks is rarely done in production services since the contention for shared resources can cause latency spikes that violate the service-level objectives of latency-sensitive tasks. The resulting under-utilization hurts both the affordability and energy-efficiency of large-scale datacenters. With technology scaling slowing down, it becomes important to address this opportunity.

We present Heracles, a feedback-based controller that enables the safe colocation of best-effort tasks alongside a latency-critical service. Heracles dynamically manages multiple hardware and software isolation mechanisms, such as CPU, memory, and network isolation, to ensure that the latency-sensitive job meets latency targets while maximizing the resources given to best-effort tasks. We evaluate Heracles using production latency-critical and batch workloads from Google and demonstrate average server utilizations of 90% without latency violations across all the load and colocation scenarios that we evaluated.

1 Introduction

Public and private cloud frameworks allow us to host an increasing number of workloads in large-scale datacenters with tens of thousands of servers. The business models for cloud services emphasize reduced infrastructure costs. Of the total cost of ownership (TCO) for modern energy-efficient datacenters, servers are the largest fraction (50-70%) [7]. Maximizing server utilization is therefore important for continued scaling.

Until recently, scaling from Moore's law provided higher compute per dollar with every server generation, allowing datacenters to scale without raising the cost. However, with several imminent challenges in technology scaling [21, 25], alternate approaches are needed. Some efforts seek to reduce the server cost through balanced designs or cost-effective components [31, 48, 42]. An orthogonal approach is to improve the return on investment and utility of datacenters by raising server utilization. Low utilization negatively impacts both operational and capital components of cost efficiency. Energy proportionality can reduce operational expenses at low utilization [6, 47]. But, to amortize the much larger capital expenses, an increased emphasis on the *effective use of server resources* is warranted.

Several studies have established that the average server utilization in most datacenters is low, ranging between 10% and 50% [14, 74, 66, 7, 19, 13]. A primary reason for the low utilization is the popularity of latency-critical (LC) services such as social media, search engines, software-as-a-service, online maps, webmail, machine translation, online shopping and advertising. These user-facing services are typically scaled across thousands of servers and access distributed state stored in memory or Flash across these servers. While their load varies significantly due to diurnal patterns and unpredictable spikes in user accesses, it is difficult to consolidate load on a subset of highly utilized servers because the application state does not fit in a small number of servers and moving state is expensive. The cost of such underutilization can be significant. For instance, Google websearch servers often have an average idleness of 30% over a 24 hour period [47]. For a hypothetical cluster of 10,000 servers, this idleness translates to a wasted capacity of 3,000 servers.

A promising way to improve efficiency is to launch best-effort batch (BE) tasks on the same servers and exploit any resources underutilized by LC workloads [52, 51, 18]. Batch analytics frameworks can generate numerous BE tasks and derive significant value even if these tasks are occasionally deferred or restarted [19, 10, 13, 16]. The main challenge of this approach is interference between colocated workloads on shared resources such as caches, memory, I/O channels, and network links. LC tasks operate with strict service level objectives (SLOs) on tail latency, and even small amounts of interference can cause significant SLO violations [51, 54, 39]. Hence, some of the past work on workload colocation focused only on throughput workloads [58, 15]. More recent systems predict or detect when a LC task suffers significant interference from the colocated tasks, and *avoid or terminate* the colocation [75, 60, 19, 50, 51, 81]. These systems protect LC workloads, but reduce the opportunities for higher utilization through colocation.

Recently introduced hardware features for cache isolation and fine-grained power control allow us to improve colocation. This work aims to enable aggressive colocation of LC workloads and BE jobs by automatically coordinating multiple hardware and software isolation mechanisms in modern servers. We focus on two hardware mechanisms, shared cache partitioning and fine-grained power/frequency settings, and two software mechanisms, core/thread scheduling and network traffic control. Our goal is to eliminate SLO violations at all levels of load for the LC job while maximizing the throughput for BE tasks.

There are several challenges towards this goal. First, we must carefully share each individual resource; conservative allocation will minimize the throughput for BE tasks, while optimistic allocation will lead to SLO violations for the LC tasks. Second, the performance of both types of tasks depends on multiple resources, which leads to a large allocation space that must be

ISCA '15, June 13 - 17, 2015, Portland, OR, USA
 ISBN 978-1-4503-3402-0/15/06$15.00
DOI: http://dx.doi.org/10.1145/2749469.2749475

explored in real-time as load changes. Finally, there are non-obvious interactions between isolated and non-isolated resources in modern servers. For instance, increasing the cache allocation for a LC task to avoid evictions of hot data may create memory bandwidth interference due to the increased misses for BE tasks.

We present *Heracles*[1], a real-time, dynamic controller that manages four hardware and software isolation mechanisms in a coordinated fashion to maintain the SLO for a LC job. Compared to existing systems [80, 51, 19] that *prevent* colocation of interfering workloads, *Heracles enables* a LC task to be colocated with any BE job. It guarantees that the LC workload receives just enough of each shared resource to meet its SLO, thereby maximizing the utility from the BE task. Using online monitoring and some offline profiling information for LC jobs, *Heracles* identifies when shared resources become saturated and are likely to cause SLO violations and configures the appropriate isolation mechanism to proactively prevent that from happening.

The specific contributions of this work are the following. First, we characterize the impact of interference on shared resources for a set of production, latency-critical workloads at Google, including websearch, an online machine learning clustering algorithm, and an in-memory key-value store. We show that the impact of interference is non-uniform and workload dependent, thus precluding the possibility of static resource partitioning within a server. Next, we design *Heracles* and show that: a) coordinated management of multiple isolation mechanisms is key to achieving high utilization without SLO violations; b) carefully separating interference into independent subproblems is effective at reducing the complexity of the dynamic control problem; and c) a local, real-time controller that monitors latency in each server is sufficient. We evaluate *Heracles* on production Google servers by using it to colocate production LC and BE tasks . We show that *Heracles* achieves an effective machine utilization of 90% averaged across all colocation combinations and loads for the LC tasks while meeting the latency SLOs. *Heracles* also improves throughput/TCO by 15% to 300%, depending on the initial average utilization of the datacenter. Finally, we establish the need for hardware mechanisms to monitor and isolate DRAM bandwidth, which can improve *Heracles*' accuracy and eliminate the need for offline information.

To the best of our knowledge, this is the first study to make coordinated use of new and existing isolation mechanisms in a real-time controller to demonstrate significant improvements in efficiency for production systems running LC services.

2 Shared Resource Interference

When two or more workloads execute concurrently on a server, they compete for shared resources. This section reviews the major sources of interference, the available isolation mechanisms, and the motivation for dynamic management.

The primary shared resource in the server are the **cores** in the one or more CPU sockets. We cannot simply *statically* partition cores between the LC and BE tasks using mechanisms such as `cgroups cpuset` [55]. When user-facing services such as search face a load spike, they need all available cores to meet throughput demands without latency SLO violations. Similarly, we cannot simply assign high priority to LC tasks and rely on OS-level scheduling of cores between tasks. Common scheduling algorithms such as Linux's completely fair scheduler (CFS) have vulnerabilities that lead to frequent SLO violations when LC tasks are colocated with BE tasks [39]. Real-time scheduling algorithms (e.g., `SCHED_FIFO`) are not work-preserving and lead to lower utilization. The availability of HyperThreads in Intel cores leads to further complications, as a HyperThread executing a BE task can interfere with a LC HyperThread on instruction bandwidth, shared L1/L2 caches, and TLBs.

Numerous studies have shown that uncontrolled interference on the shared **last-level cache (LLC)** can be detrimental for colocated tasks [68, 50, 19, 22, 39]. To address this issue, Intel has recently introduced LLC cache partitioning in server chips. This functionality is called *Cache Allocation Technology (CAT)*, and it enables way-partitioning of a highly-associative LLC into several subsets of smaller associativity [3]. Cores assigned to one subset can only allocate cache lines in their subset on refills, but are allowed to hit in any part of the LLC. It is already well understood that, even when the colocation is between throughput tasks, it is best to dynamically manage cache partitioning using either hardware [30, 64, 15] or software [58, 43] techniques. In the presence of user-facing workloads, dynamic management is more critical as interference translates to large latency spikes [39]. It is also more challenging as the cache footprint of user-facing workloads changes with load [36].

Most important LC services operate on large datasets that do not fit in on-chip caches. Hence, they put pressure on DRAM bandwidth at high loads and are sensitive to **DRAM bandwidth** interference. Despite significant research on memory bandwidth isolation [30, 56, 32, 59], there are no hardware isolation mechanisms in commercially available chips. In multi-socket servers, one can isolate workloads across NUMA channels [9, 73], but this approach constrains DRAM capacity allocation and address interleaving. The lack of hardware support for memory bandwidth isolation complicates and constrains the efficiency of any system that dynamically manages workload colocation.

Datacenter workloads are scale-out applications that generate **network traffic**. Many datacenters use rich topologies with sufficient bisection bandwidth to avoid routing congestion in the fabric [28, 4]. There are also several networking protocols that prioritize short messages for LC tasks over large messages for BE tasks [5, 76]. Within a server, interference can occur both in the incoming and outgoing direction of the network link. If a BE task causes incast interference, we can throttle its core allocation until networking flow-control mechanisms trigger [62]. In the outgoing direction, we can use traffic control mechanisms in operating systems like Linux to provide bandwidth guarantees to LC tasks and to prioritize their messages ahead of those from BE tasks [12]. Traffic control must be managed dynamically as bandwidth requirements vary with load. Static priorities can cause underutilization and starvation [61]. Similar traffic control can be applied to solid-state storage devices [69].

Power is an additional source of interference between colocated tasks. All modern multi-core chips have some form of dynamic overclocking, such as Turbo Boost in Intel chips and Turbo Core in AMD chips. These techniques opportunistically raise the operating frequency of the processor chip higher than the nominal frequency in the presence of power headroom. Thus, the clock frequency for the cores used by a LC task depends not

[1] The mythical hero that killed the multi-headed monster, Lernaean Hydra.

just on its own load, but also on the intensity of any BE task running on the same socket. In other words, the performance of LC tasks can suffer from unexpected drops in frequency due to colocated tasks. This interference can be mitigated with per-core dynamic voltage frequency scaling, as cores running BE tasks can have their frequency decreased to ensure that the LC jobs maintain a guaranteed frequency. A static policy would run all BE jobs at minimum frequency, thus ensuring that the LC tasks are not power-limited. However, this approach severely penalizes the vast majority of BE tasks. Most BE jobs do not have the profile of a power virus[2] and LC tasks only need the additional frequency boost during periods of high load. Thus, a dynamic solution that adjusts the allocation of power between cores is needed to ensure that LC cores run at a guaranteed minimum frequency while maximizing the frequency of cores for BE tasks.

A major challenge with colocation is **cross-resource interactions**. A BE task can cause interference in all the shared resources discussed. Similarly, many LC tasks are sensitive to interference on multiple resources. Therefore, it is not sufficient to manage one source of interference: all potential sources need to be monitored and carefully isolated if need be. In addition, interference sources interact with each other. For example, LLC contention causes both types of tasks to require more DRAM bandwidth, also creating a DRAM bandwidth bottleneck. Similarly, a task that notices network congestion may attempt to use compression, causing core and power contention. In theory, the number of possible interactions scales with the square of the number of interference sources, making this a very difficult problem.

3 Interference Characterization & Analysis

This section characterizes the impact of interference on shared resources for latency-critical services.

3.1 Latency-critical Workloads

We use three Google production latency-critical workloads.

websearch is the query serving portion of a production web search service. It is a scale-out workload that provides high throughput with a strict latency SLO by using a large fan-out to thousands of leaf nodes that process each query on their shard of the search index. The SLO for leaf nodes is in the tens of milliseconds for the 99%-ile latency. Load for *websearch* is generated using an anonymized trace of real user queries.

websearch has high memory footprint as it serves shards of the search index stored in DRAM. It also has moderate DRAM bandwidth requirements (40% of available bandwidth at 100% load), as most index accesses miss in the LLC. However, there is a small but significant working set of instructions and data in the hot path. Also, *websearch* is fairly compute intensive, as it needs to score and sort search hits. However, it does not consume a significant amount of network bandwidth. For this study, we reserve a small fraction of DRAM on search servers to enable colocation of BE workloads with *websearch*.

ml_cluster is a standalone service that performs real-time text clustering using machine-learning techniques. Several Google services use *ml_cluster* to assign a cluster to a snippet of text. *ml_cluster* performs this task by locating the closest clusters for the text in a model that was previously learned offline. This model is kept in main memory for performance reasons. The SLO for *ml_cluster* is a 95%-ile latency guarantee of tens of milliseconds. *ml_cluster* is exercised using an anonymized trace of requests captured from production services.

Compared to *websearch*, *ml_cluster* is more memory bandwidth intensive (with 60% DRAM bandwidth usage at peak) but slightly less compute intensive (lower CPU power usage overall). It has low network bandwidth requirements. An interesting property of *ml_cluster* is that each request has a very small cache footprint, but, in the presence of many outstanding requests, this translates into a large amount of cache pressure that spills over to DRAM. This is reflected in our analysis as a super-linear growth in DRAM bandwidth use for *ml_cluster* versus load.

memkeyval is an in-memory key-value store, similar to *memcached* [2]. *memkeyval* is used as a caching service in the backends of several Google web services. Other large-scale web services, such as Facebook and Twitter, use *memcached* extensively. *memkeyval* has significantly less processing per request compared to *websearch*, leading to extremely high throughput in the order of hundreds of thousands of requests per second at peak. Since each request is processed quickly, the SLO latency is very low, in the few hundreds of microseconds for the 99%-ile latency. Load generation for *memkeyval* uses an anonymized trace of requests captured from production services.

At peak load, *memkeyval* is network bandwidth limited. Despite the small amount of network protocol processing done per request, the high request rate makes *memkeyval* compute-bound. In contrast, DRAM bandwidth requirements are low (20% DRAM bandwidth utilization at max load), as requests simply retrieve values from DRAM and put the response on the wire. *memkeyval* has both a static working set in the LLC for instructions, as well as a per-request data working set.

3.2 Characterization Methodology

To understand their sensitivity to interference on shared resources, we ran each of the three LC workloads with a synthetic benchmark that stresses each resource in isolation. While these are single node experiments, there can still be significant network traffic as the load is generated remotely. We repeated the characterization at various load points for the LC jobs and recorded the impact of the colocation on tail latency. We used production Google servers with dual-socket Intel Xeons based on the Haswell architecture. Each CPU has a high core-count, with a nominal frequency of 2.3GHz and 2.5MB of LLC per core. The chips have hardware support for way-partitioning of the LLC.

We performed the following characterization experiments:

Cores: As we discussed in §2, we cannot share a logical core (a single HyperThread) between a LC and a BE task because OS scheduling can introduce latency spikes in the order of tens of milliseconds [39]. Hence, we focus on the potential of using separate HyperThreads that run pinned on the same physical core. We characterize the impact of a colocated HyperThread that implements a tight spinloop on the LC task. This experiment captures a *lower bound* of HyperThread interference. A more compute or memory intensive microbenchmark would antagonize the LC HyperThread for more core resources (e.g., execution units) and space in the private caches (L1 and L2). Hence, if this experiment shows high impact on tail latency, we can conclude that

[2] A computation that maximizes activity and power consumption of a core.

core sharing through HyperThreads is not a practical option.

LLC: The interference impact of LLC antagonists is measured by pinning the LC workload to enough cores to satisfy its SLO at the specific load and pinning a cache antagonist that streams through a large data array on the remaining cores of the socket. We use several array sizes that take up a quarter, half, and almost all of the LLC and denote these configurations as LLC small, medium, and big respectively.

DRAM bandwidth: The impact of DRAM bandwidth interference is characterized in a similar fashion to LLC interference, using a significantly larger array for streaming. We use *numactl* to ensure that the DRAM antagonist and the LC task are placed on the same socket(s) and that all memory channels are stressed.

Network traffic: We use *iperf*, an open source TCP streaming benchmark [1], to saturate the network transmit (outgoing) bandwidth. All cores except for one are given to the LC workload. Since the LC workloads we consider serve request from multiple clients connecting to the service they provide, we generate interference in the form of many low-bandwidth "mice" flows. Network interference can also be generated using a few "elephant" flows. However, such flows can be effectively throttled by TCP congestion control [11], while the many "mice" flows of the LC workload will not be impacted.

Power: To characterize the latency impact of a power antagonist, the same division of cores is used as in the cases of generating LLC and DRAM interference. Instead of running a memory access antagonist, a CPU power virus is used. The power virus is designed such that it stresses all the components of the core, leading to high power draw and lower CPU core frequencies.

OS Isolation: For completeness, we evaluate the overall impact of running a BE task along with a LC workload using only the isolation mechanisms available in the OS. Namely, we execute the two workloads in separate Linux containers and set the BE workload to be low priority. The scheduling policy is enforced by CFS using the `shares` parameter, where the BE task receives very few `shares` compared to the LC workload. No other isolation mechanisms are used in this case. The BE task is the Google *brain* workload [38, 67], which we will describe further in §5.1.

3.3 Interference Analysis

Figure 1 presents the impact of the interference microbenchmarks on the tail latency of the three LC workloads. Each row in the table shows tail latency at a certain load for the LC workload when colocated with the corresponding microbenchmark. The interference impact is acceptable if and only if the tail latency is less than 100% of the target SLO. We color-code red/yellow all cases where SLO latency is violated.

By observing the rows for *brain*, we immediately notice that current OS isolation mechanisms are inadequate for colocating LC tasks with BE tasks. Even at low loads, the BE task creates sufficient pressure on shared resources to lead to SLO violations for all three workloads. A large contributor to this is that the OS allows both workloads to run on the same core and even the same HyperThread, further compounding the interference. Tail latency eventually goes above 300% of SLO latency. Proposed interference-aware cluster managers, such as Paragon [18] and Bubble-Up [51], would disallow these colations. To enable aggressive task colocation, not only do we need to disallow different workloads on the same core or HyperThread, we also need to use stronger isolation mechanisms.

The sensitivity of LC tasks to interference on individual shared resources varies. For instance, *memkeyval* is quite sensitive to network interference, while *websearch* and *ml_cluster* are not affected at all. *websearch* is uniformly insensitive to small and medium amounts of LLC interference, while the same cannot be said for *memkeyval* or *ml_cluster*. Furthermore, the impact of interference changes depending on the load: *ml_cluster* can tolerate medium amounts of LLC interference at loads <50% but is heavily impacted at higher loads. These observations motivate the need for dynamic management of isolation mechanisms in order to adapt to differences across varying loads and different workloads. Any static policy would be either too conservative (missing opportunities for colocation) or overly optimistic (leading to SLO violations).

We now discuss each LC workload separately, in order to understand their particular resource requirements.

websearch: This workload has a small footprint and LLC (small) and LLC (med) interference do not impact its tail latency. Nevertheless, the impact is significant with LLC (big) interference. The degradation is caused by two factors. First, the inclusive nature of the LLC in this particular chip means that high LLC interference leads to misses in the working set of instructions. Second, contention for the LLC causes significant DRAM pressure as well. *websearch* is particularly sensitive to interference caused by DRAM bandwidth saturation. As the load of *websearch* increases, the impact of LLC and DRAM interference decreases. At higher loads, *websearch* uses more cores while the interference generator is given fewer cores. Thus, *websearch* can defend its share of resources better.

websearch is moderately impacted by HyperThread interference until high loads. This indicates that the core has sufficient instruction issue bandwidth for both the spinloop and the *websearch* until around 80% load. Since the spinloop only accesses registers, it doesn't cause interference in the L1 or L2 caches. However, since the HyperThread antagonist has the smallest possible effect, more intensive antagonists will cause far larger performance problems. Thus, HyperThread interference in practice should be avoided. Power interference has a significant impact on *websearch* at lower utilization, as more cores are executing the power virus. As expected, the network antagonist does not impact *websearch*, due to *websearch*'s low bandwidth needs.

ml_cluster *ml_cluster* is sensitive to LLC interference of smaller size, due to the small but significant per-request working set. This manifests itself as a large jump in latency at 75% load for LLC (small) and 50% load for LLC (medium). With larger LLC interference, *ml_cluster* experiences major latency degradation. *ml_cluster* is also sensitive to DRAM bandwidth interference, primarily at lower loads (see explanation for *websearch*). *ml_cluster* is moderately resistant to HyperThread interference until high loads, suggesting that it only reaches high instruction issue rates at high loads. Power interference has a lesser impact on *ml_cluster* since it is less compute intensive than *websearch*. Finally, *ml_cluster* is not impacted at all by network interference.

memkeyval: Due to its significantly stricter latency SLO, *memkeyval* is sensitive to all types of interference. At high load, *memkeyval* becomes sensitive even to small LLC interference as

websearch

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
LLC (small)	134%	103%	96%	96%	109%	102%	100%	96%	96%	104%	99%	100%	101%	100%	104%	103%	104%	103%	99%
LLC (med)	152%	106%	99%	99%	116%	111%	109%	103%	105%	116%	109%	108%	107%	110%	123%	125%	114%	111%	101%
LLC (big)	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	264%	222%	123%	102%
DRAM	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	270%	228%	122%	103%
HyperThread	81%	109%	106%	106%	104%	113%	106%	114%	113%	105%	114%	117%	118%	119%	122%	136%	>300%	>300%	>300%
CPU power	190%	124%	110%	107%	134%	115%	106%	108%	102%	114%	107%	105%	104%	101%	105%	100%	98%	99%	97%
Network	35%	35%	36%	36%	36%	36%	36%	37%	37%	38%	39%	41%	44%	48%	51%	55%	58%	64%	95%
brain	158%	165%	157%	173%	160%	168%	180%	230%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%

ml_cluster

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
LLC (small)	101%	88%	99%	84%	91%	110%	96%	93%	100%	216%	117%	106%	119%	105%	182%	206%	109%	202%	203%
LLC (med)	98%	88%	102%	91%	112%	115%	105%	104%	111%	>300%	282%	212%	237%	220%	220%	212%	215%	205%	201%
LLC (big)	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	276%	250%	223%	214%	206%
DRAM	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	287%	230%	223%	211%
HyperThread	113%	109%	110%	111%	104%	100%	97%	107%	111%	112%	114%	114%	114%	119%	121%	130%	259%	262%	262%
CPU power	112%	101%	97%	89%	91%	86%	89%	90%	89%	92%	91%	90%	89%	89%	90%	92%	94%	97%	106%
Network	57%	56%	58%	60%	58%	58%	58%	58%	59%	59%	59%	59%	59%	63%	63%	67%	76%	89%	113%
brain	151%	149%	174%	189%	193%	202%	209%	217%	225%	239%	>300%	>300%	279%	>300%	>300%	>300%	>300%	>300%	>300%

memkeyval

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
LLC (small)	115%	88%	88%	91%	99%	101%	79%	91%	97%	101%	135%	138%	148%	140%	134%	150%	114%	78%	70%
LLC (med)	209%	148%	159%	107%	207%	119%	96%	108%	117%	138%	170%	230%	182%	181%	167%	162%	144%	100%	104%
LLC (big)	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	280%	225%	222%	170%	79%	85%
DRAM	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	252%	234%	199%	103%	100%
HyperThread	26%	31%	32%	32%	32%	32%	33%	35%	39%	43%	48%	51%	56%	62%	81%	119%	116%	153%	>300%
CPU power	192%	277%	237%	294%	>300%	>300%	219%	>300%	292%	224%	>300%	252%	227%	193%	163%	167%	122%	82%	123%
Network	27%	28%	28%	29%	29%	27%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%
brain	197%	232%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%

Each entry is color-coded as follows: 140% is ≥120%, 110% is between 100% and 120%, and 65% is ≤100%.

Figure 1. Impact of interference on shared resources on websearch, ml_cluster, and memkeyval. Each row is an antagonist and each column is a load point for the workload. The values are latencies, normalized to the SLO latency.

the small per-request working sets add up. When faced with medium LLC interference, there are two latency peaks. The first peak at low load is caused by the antagonist removing instructions from the cache. When *memkeyval* obtains enough cores at high load, it avoids these evictions. The second peak is at higher loads, when the antagonist interferes with the per-request working set. At high levels of LLC interference, *memkeyval* is unable to meet its SLO. Even though *memkeyval* has low DRAM bandwidth requirements, it is strongly affected by a DRAM streaming antagonist. Ironically, the few memory requests from *memkeyval* are overwhelmed by the DRAM antagonist.

memkeyval is not sensitive to the HyperThread antagonist except at high loads. In contrast, it is very sensitive to the power antagonist, as it is compute-bound. *memkeyval* does consume a large amount of network bandwidth, and thus is highly susceptible to competing network flows. Even at small loads, it is completely overrun by the many small "mice" flows of the antagonist and is unable to meet its SLO.

4 Heracles Design

We have established the need for isolation mechanisms beyond OS-level scheduling and for a dynamic controller that manages resource sharing between LC and BE tasks. *Heracles* is a dynamic, feedback-based controller that manages in real-time four hardware and software mechanisms in order to isolate colocated workloads. *Heracles* implements an *iso-latency* policy [47], namely that it can increase resource efficiency as long as the SLO is being met. This policy allows for increasing server utilization through tolerating some interference caused by colocation, as long as the the difference between the SLO latency target for the LC workload and the actual latency observed (latency slack) is positive. In its current version, *Heracles* manages one LC workload with many BE tasks. Since BE tasks are abundant, this is sufficient to raise utilization in many datacenters. We leave colocation of multiple LC workloads to future work.

4.1 Isolation Mechanisms

Heracles manages 4 mechanisms to mitigate interference.

For **core isolation**, *Heracles* uses Linux's `cpuset cgroups` to pin the LC workload to one set of cores and BE tasks to another set (software mechanism) [55]. This mechanism is necessary, since in §3 we showed that core sharing is detrimental to latency SLO. Moreover, the number of cores per server is increasing, making core segregation finer-grained. The allocation of cores to tasks is done dynamically. The speed of core (re)allocation is limited by how fast Linux can migrate tasks to other cores, typically in the tens of milliseconds.

For **LLC isolation**, *Heracles* uses the Cache Allocation Technology (CAT) available in recent Intel chips (hardware mechanism) [3]. CAT implements way-partitioning of the shared LLC. In a highly-associative LLC, this allows us to define non-overlapping partitions at the granularity of a few percent of the total LLC capacity. We use one partition for the LC workload and a second partition for all BE tasks. Partition sizes can be adjusted dynamically by programming model specific registers (MSRs), with changes taking effect in a few milliseconds.

There are no commercially available DRAM bandwidth isolation mechanisms. We enforce DRAM bandwidth limits in the following manner: we implement a software monitor that periodically tracks the total bandwidth usage through performance counters and estimates the bandwidth used by the LC and BE jobs. If the LC workload does not receive sufficient bandwidth, *Heracles* scales down the number of cores that BE jobs use. We

discuss the limitations of this coarse-grained approach in §4.2.

For **power isolation**, *Heracles* uses CPU frequency monitoring, Running Average Power Limit (RAPL), and per-core DVFS (hardware features) [3, 37]. RAPL is used to monitor CPU power at the per-socket level, while per-core DVFS is used to redistribute power amongst cores. Per-core DVFS setting changes go into effect within a few milliseconds. The frequency steps are in 100MHz and span the entire operating frequency range of the processor, including Turbo Boost frequencies.

For **network traffic isolation**, *Heracles* uses Linux traffic control (software mechanism). Specifically we use the `qdisc` [12] scheduler with hierarchical token bucket queueing discipline (HTB) to enforce bandwidth limits for outgoing traffic from the BE tasks. The bandwidth limits are set by limiting the maximum traffic burst rate for the BE jobs (`ceil` parameter in HTB parlance). The LC job does not have any limits set on it. HTB can be updated very frequently, with the new bandwidth limits taking effect in less than hundreds of milliseconds. Managing ingress network interference has been examined in numerous previous work and is outside the scope of this work [33].

4.2 Design Approach

Each hardware or software isolation mechanism allows reasonably precise control of an individual resource. Given that, the controller must dynamically solve the high dimensional problem of finding the right settings for all these mechanisms at any load for the LC workload and any set of BE tasks. *Heracles* solves this as an *optimization problem*, where the *objective* is to maximize utilization with the *constraint* that the SLO must be met.

Heracles reduces the optimization complexity by decoupling interference sources. The key insight that enables this reduction is that *interference is problematic only when a shared resource becomes saturated*, i.e. its utilization is so high that latency problems occur. This insight is derived by the analysis in §3: the antagonists do not cause significant SLO violations until an inflection point, at which point the tail latency degrades extremely rapidly. Hence, if *Heracles* can prevent any shared resource from saturating, then *it can decompose the high-dimensional optimization problem into many smaller and independent problems of one or two dimensions each*. Then each sub-problem can be solved using sound optimization methods, such as gradient descent.

Since *Heracles* must ensure that the target SLO is met for the LC workload, it continuously monitors latency and latency slack and uses both as key inputs in its decisions. When the latency slack is large, *Heracles* treats this as a signal that it is safe to be more aggressive with colocation; conversely, when the slack is small, it should back off to avoid an SLO violation. *Heracles* also monitors the load (queries per second), and during periods of high load, it disables colocation due to a high risk of SLO violations. Previous work has shown that indirect performance metrics, such as CPU utilization, are insufficient to guarantee that the SLO is met [47].

Ideally, *Heracles* should require no offline information other than SLO targets. Unfortunately, one shortcoming of current hardware makes this difficult. The Intel chips we used do not provide accurate mechanisms for measuring (or limiting) DRAM bandwidth usage at a per-core granularity. To understand how *Heracles*' decisions affect the DRAM bandwidth usage of

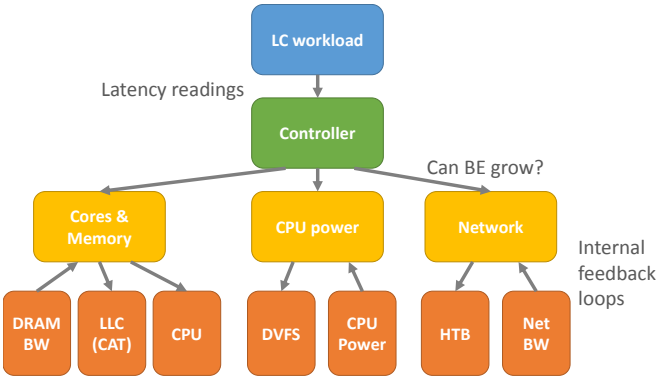

Figure 2. The system diagram of *Heracles*.

latency-sensitive and BE tasks and to manage bandwidth saturation, we require some offline information. Specifically, *Heracles* uses an offline model that describes the DRAM bandwidth used by the latency-sensitive workloads at various loads, core, and LLC allocations. We verified that this model needs to be regenerated only when there are significant changes in the workload structure and that small deviations are fine. There is no need for any offline profiling of the BE tasks, which can vary widely compared to the better managed and understood LC workloads. There is also no need for offline analysis of interactions between latency-sensitive and best effort tasks. Once we have hardware support for per-core DRAM bandwidth accounting [30], we can eliminate this offline model.

4.3 Heracles Controller

Heracles runs as a separate instance on each server, managing the local interactions between the LC and BE jobs. As shown in Figure 2, it is organized as three subcontrollers (cores & memory, power, network traffic) coordinated by a top-level controller. The subcontrollers operate fairly independently of each other and ensure that their respective shared resources are not saturated.

Top-level controller: The pseudo-code for the controller is shown in Algorithm 1. The controller polls the tail latency and load of the LC workload every 15 seconds. This allows for sufficient queries to calculate statistically meaningful tail latencies. If the load for the LC workload exceeds 85% of its peak on the server, the controller disables the execution of BE workloads. This empirical safeguard avoids the difficulties of latency management on highly utilized systems for minor gains in utilization. For hysteresis purposes, BE execution is enabled when the load drops below 80%. BE execution is also disabled when the latency slack, the difference between the SLO target and the current measured tail latency, is negative. This typically happens when there is a sharp spike in load for the latency-sensitive workload. We give all resources to the latency critical workload for a while (e.g., 5 minutes) before attempting colocation again. The constants used here were determined through empirical tuning.

When these two safeguards are not active, the controller uses slack to guide the subcontrollers in providing resources to BE tasks. If slack is less than 10%, the subcontrollers are instructed to disallow growth for BE tasks in order to maintain a safety margin. If slack drops below 5%, the subcontroller for cores is instructed to switch cores from BE tasks to the LC workload. This improves the latency of the LC workload and reduces the

```
while True:
    latency=PollLCAppLatency()
    load=PollLCAppLoad()
    slack=(target-latency)/target
    if slack<0:
        DisableBE()
        EnterCooldown()
    elif load>0.85:
        DisableBE()
    elif load<0.80:
        EnableBE()
    elif slack<0.10:
        DisallowBEGrowth()
        if slack<0.05:
            be_cores.Remove(be_cores.Size()-2)
    sleep(15)
```

Algorithm 1: High-level controller.

ability of the BE job to cause interference on any resources. If slack is above 10%, the subcontrollers are instructed to allow BE tasks to acquire a larger share of system resources. Each subcontroller makes allocation decisions independently, provided of course that its resources are not saturated.

Core & memory subcontroller: *Heracles* uses a single subcontroller for core and cache allocation due to the strong coupling between core count, LLC needs, and memory bandwidth needs. If there was a direct way to isolate memory bandwidth, we would use independent controllers. The pseudo-code for this subcontroller is shown in Algorithm 2. Its output is the allocation of cores and LLC to the LC and BE jobs (2 dimensions).

The first constraint for the subcontroller is to avoid memory bandwidth saturation. The DRAM controllers provide registers that track bandwidth usage, making it easy to detect when they reach 90% of peak streaming DRAM bandwidth. In this case, the subcontroller removes as many cores as needed from BE tasks to avoid saturation. *Heracles* estimates the bandwidth usage of each BE task using a model of bandwidth needs for the LC workload and a set of hardware counters that are proportional to the per-core memory traffic to the NUMA-local memory controllers. For the latter counters to be useful, we limit each BE task to a single socket for both cores and memory allocations using Linux `numactl`. Different BE jobs can run on either socket and LC workloads can span across sockets for cores and memory.

When the top-level controller signals BE growth and there is no DRAM bandwidth saturation, the subcontroller uses gradient descent to find the maximum number of cores and cache partitions that can be given to BE tasks. Offline analysis of LC applications (Figure 3) shows that their performance is a convex function of core and cache resources, thus guaranteeing that gradient descent will find a global optimum. We perform the gradient descent in one dimension at a time, switching between increasing the cores and increasing the cache given to BE tasks. Initially, a BE job is given one core and 10% of the LLC and starts in the *GROW_LLC* phase. Its LLC allocation is increased as long as the LC workload meets its SLO, bandwidth saturation is avoided, and the BE task benefits. The next phase (*GROW_CORES*) grows the number of cores for the BE job. *Heracles* will reassign cores from the LC to the BE job one at a

```
def PredictedTotalBW():
    return LcBwModel()+BeBw()+bw_derivative
while True:
    MeasureDRAMBw()
    if total_bw>DRAM_LIMIT:
        overage=total_bw-DRAM_LIMIT
        be_cores.Remove(overage/BeBwPerCore())
        continue
    if not CanGrowBE():
        continue
    if state==GROW_LLC:
        if PredictedTotalBW()>DRAM_LIMIT:
            state=GROW_CORES
        else:
            GrowCacheForBE()
            MeasureDRAMBw()
            if bw_derivative>=0:
                Rollback()
                state=GROW_CORES
            if not BeBenefit():
                state=GROW_CORES
    elif state==GROW_CORES:
        needed=LcBwModel()+BeBw()+BeBwPerCore()
        if needed>DRAM_LIMIT:
            state=GROW_LLC
        elif slack>0.10:
            be_cores.Add(1)
    sleep(2)
```

Algorithm 2: Core & memory sub-controller.

time, each time checking for DRAM bandwidth saturation and SLO violations for the LC workload. If bandwidth saturation occurs first, the subcontroller will return to the *GROW_LLC* phase. The process repeats until an optimal configuration has been converged upon. The search also terminates on a signal from the top-level controller indicating the end to growth or the disabling of BE jobs. The typical convergence time is about 30 seconds.

During gradient descent, the subcontroller must avoid trying suboptimal allocations that will either trigger DRAM bandwidth saturation or a signal from the top-level controller to disable BE tasks. To estimate the DRAM bandwidth usage of an allocation prior to trying it, the subcontroller uses the derivative of the DRAM bandwidth from the last reallocation of cache or cores. *Heracles* estimates whether it is close to an SLO violation for the LC task based on the amount of latency slack.

Power subcontroller: The simple subcontroller described in Algorithm 3 ensures that there is sufficient power slack to run the LC workload at a minimum guaranteed frequency. This frequency is determined by measuring the frequency used when the LC workload runs alone at full load. *Heracles* uses RAPL to determine the operating power of the CPU and its maximum design power, or thermal dissipation power (TDP). It also uses CPU frequency monitoring facilities on each core. When the operating power is close to the TDP **and** the frequency of the cores running the LC workload is too low, it uses per-core DVFS to lower the frequency of cores running BE tasks in order to shift the power budget to cores running LC tasks. Both conditions must be met in order to avoid confusion when the LC cores enter active-idle

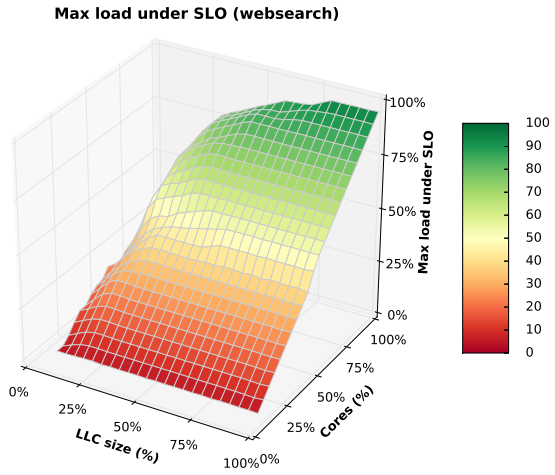

Figure 3. Characterization of *websearch* showing that its performance is a convex function of cores and LLC.

```
1 while True:
2     power=PollRAPL()
3     ls_freq=PollFrequency(ls_cores)
4     if power>0.90*TDP and ls_freq<guaranteed:
5         LowerFrequency(be_cores)
6     elif power<=0.90*TDP and ls_freq>=guaranteed:
7         IncreaseFrequency(be_cores)
8     sleep(2)
```

Algorithm 3: CPU power sub-controller.

```
1 while True:
2     ls_bw=GetLCTxBandwidth()
3     be_bw=LINK_RATE-ls_bw-max(0.05*LINK_RATE,
      0.10*ls_bw)
4     SetBETxBandwidth(be_bw)
5     sleep(1)
```

Algorithm 4: Network sub-controller.

modes, which also tends to lower frequency readings. If there is sufficient operating power headroom, *Heracles* will increase the frequency limit for the BE cores in order to maximize their performance. The control loop runs independently for each of the two sockets and has a cycle time of two seconds.

Network subcontroller: This subcontroller prevents saturation of network transmit bandwidth as shown in Algorithm 4. It monitors the total egress bandwidth of flows associated with the LC workload (*LCBandwidth*) and sets the total bandwidth limit of all other flows as $LinkRate - LCBandwidth - max(0.05LinkRate, 0.10LCBandwidth)$. A small headroom of 10% of the current *LCBandwidth* or 5% of the LinkRate is added into the reservation for the LC workload in order to handle spikes. The bandwidth limit is enforced via HTB qdiscs in the Linux kernel. This control loop is run once every second, which provides sufficient time for the bandwidth enforcer to settle.

5 Heracles Evaluation

5.1 Methodology

We evaluated *Heracles* with the three production, latency-critical workloads from Google analyzed in §3. We first performed experiments with *Heracles* on a single leaf server, introducing BE tasks as we run the LC workload at different levels of load. Next, we used *Heracles* on a *websearch* cluster with tens of servers, measuring end-to-end workload latency across the fan-out tree while BE tasks are also running. In the cluster experiments, we used a load trace that represents the traffic throughout a day, capturing diurnal load variation. In all cases, we used production Google servers.

For the LC workloads we focus on SLO latency. Since the SLO is defined over 60-second windows, we report the worst-case latency that was seen during experiments. For the production batch workloads, we compute the throughput rate of the batch workload with *Heracles* and normalize it to the throughput of the batch workload running alone on a single server. We then define the **Effective Machine Utilization (EMU)** = LC Throughput + BE Throughput. Note that **Effective Machine Utilization** can be above 100% due to better binpacking of shared resources. We also report the utilization of shared resources when necessary to highlight detailed aspects of the system.

The BE workloads we use are chosen from a set containing both production batch workloads and the synthetic tasks that stress a single shared resource. The specific workloads are:

stream-LLC streams through data sized to fit in about half of the LLC and is the same as LLC (med) from §3.2. **stream-DRAM** streams through an extremely large array that cannot fit in the LLC (DRAM from the same section). We use these workloads to verify that *Heracles* is able to maximize the use of LLC partitions and avoid DRAM bandwidth saturation.

cpu_pwr is the CPU power virus from §3.2. It is used to verify that *Heracles* will redistribute power to ensure that the LC workload maintains its guaranteed frequency.

iperf is an open source network streaming benchmark used to verify that *Heracles* partitions network transmit bandwidth correctly to protect the LC workload.

brain is a Google production batch workload that performs deep learning on images for automatic labelling [38, 67]. This workload is very computationally intensive, is sensitive to LLC size, and also has high DRAM bandwidth requirements.

streetview is a production batch job that stitches together multiple images to form the panoramas for Google Street View. This workload is highly demanding on the DRAM subsystem.

5.2 Individual Server Results

Latency SLO: Figure 4 presents the impact of colocating each of the three LC workloads with BE workloads across all possible loads under the control of *Heracles*. Note that *Heracles* attempts to run as many copies of the BE task as possible and maximize the resources they receive. At all loads and in all colocation cases, there are *no SLO violations* with *Heracles*. This is true even for *brain*, a workload that even with the state-of-the-art OS isolation mechanisms would render any LC workload unusable. This validates that the controller keeps shared resources from saturating and allocates a sufficient fraction to the LC workload at any load. *Heracles* maintains a small

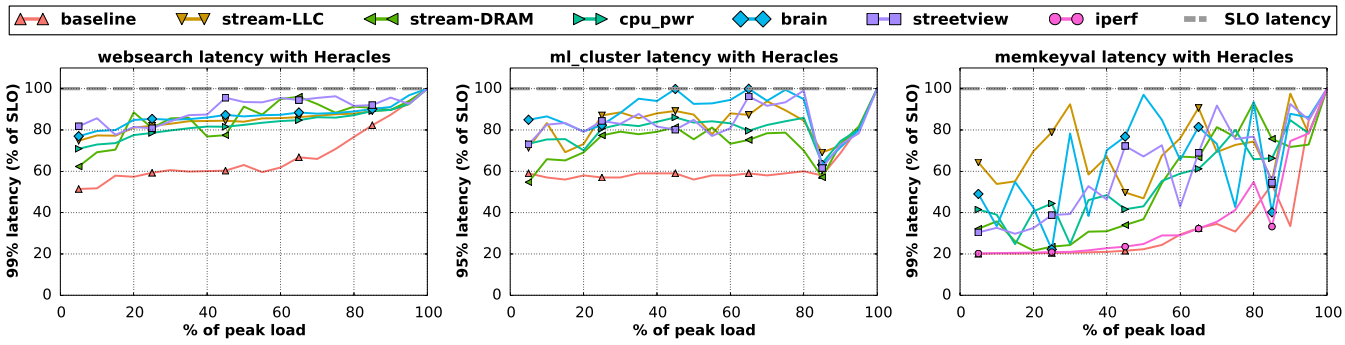

Figure 4. Latency of LC applications co-located with BE jobs under *Heracles*. For clarity we omit websearch and ml_cluster with iperf as those workloads are extremely resistant to network interference.

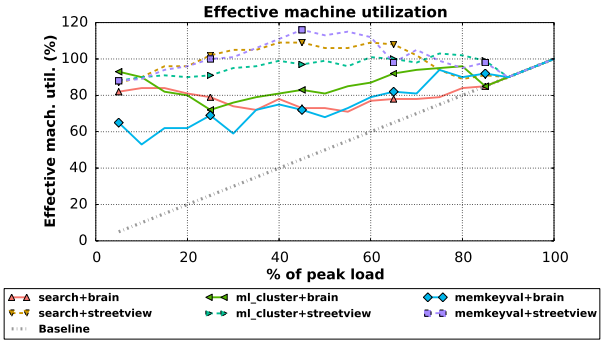

Figure 5. EMU achieved by *Heracles*.

latency slack as a guard band to avoid spikes and control instability. It also validates that local information on tail latency is sufficient for stable control for applications with milliseconds and microseconds range of SLOs. Interestingly, the *websearch* binary and shard changed between generating the offline profiling model for DRAM bandwidth and performing this experiment. Nevertheless, *Heracles* is resilient to these changes and performs well despite the somewhat outdated model.

Heracles reduces the latency slack during periods of low utilization for all workloads. For *websearch* and *ml_cluster*, the slack is cut in half, from 40% to 20%. For *memkeyval*, the reduction is much more dramatic, from a slack of 80% to 40% or less. This is because the unloaded latency of *memkeyval* is extremely small compared to the SLO latency. The high variance of the tail latency for *memkeyval* is due to the fact that its SLO is in the hundreds of microseconds, making it more sensitive to interference than the other two workloads.

Server Utilization: Figure 5 shows the EMU achieved when colocating production LC and BE tasks with *Heracles*. In all cases, we achieve significant EMU increases. When the two most CPU-intensive and power-hungry workloads are combined, *websearch* and *brain*, *Heracles* still achieves an EMU of at least 75%. When *websearch* is combined with the DRAM bandwidth intensive *streetview*, *Heracles* can extract sufficient resources for a total EMU above 100% at *websearch* loads between 25% and 70%. This is because *websearch* and *streetivew* have complementary resource requirements, where *websearch* is more compute bound and *streetview* is more DRAM bandwidth bound. The EMU results are similarly positive for *ml_cluster* and *memkeyval*. By dynamically managing multiple isolation mechanisms, *Heracles* exposes opportunities to raise EMU that would otherwise be missed with scheduling techniques that avoid interference.

Shared Resource Utilization: Figure 6 plots the utilization of shared resources (cores, power, and DRAM bandwidth) under *Heracles* control. For *memkeyval*, we include measurements of network transmit bandwidth in Figure 7.

Across the board, *Heracles* is able to correctly size the BE workloads to avoid saturating DRAM bandwidth. For the stream-LLC BE task, *Heracles* finds the correct cache partitions to decrease total DRAM bandwidth requirements for all workloads. For *ml_cluster*, with its large cache footprint, *Heracles* balances the needs of stream-LLC with *ml_cluster* effectively, with a total DRAM bandwidth slightly above the baseline. For the BE tasks with high DRAM requirements (stream-DRAM, streetview), *Heracles* only allows them to execute on a few cores to avoid saturating DRAM. This is reflected by the lower CPU utilization but high DRAM bandwidth. However, EMU is still high, as the critical resource for those workloads is not compute, but memory bandwidth.

Looking at the power utilization, *Heracles* allows significant improvements to energy efficiency. Consider the 20% load case: EMU was raised by a significant amount, from 20% to 60%-90%. However, the CPU power only increased from 60% to 80%. This translates to an energy efficiency gain of 2.3-3.4x. Overall, *Heracles* achieves significant gains in resource efficiency across all loads for the LC task without causing SLO violations.

5.3 Websearch Cluster Results

We also evaluate *Heracles* on a small minicluster for *websearch* with tens of servers as a proxy for the full-scale cluster. The cluster root fans out each user request to all leaf servers and combines their replies. The SLO latency is defined as the average latency at the root over 30 seconds, denoted as $\mu/30s$. The target SLO latency is set as $\mu/30s$ when serving 90% load in the cluster without colocated tasks. *Heracles* runs on every leaf node with a uniform 99%-ile latency target set such that the latency at the root satisfies the SLO. We use *Heracles* to execute *brain* on half of the leafs and *streetview* on the other half. *Heracles* shares the same offline model for the DRAM bandwidth needs of *websearch* across all leaves, even though each leaf has a different shard. We generate load from an anonymized, 12-hour request trace that captures the part of the daily diurnal pattern when *web-*

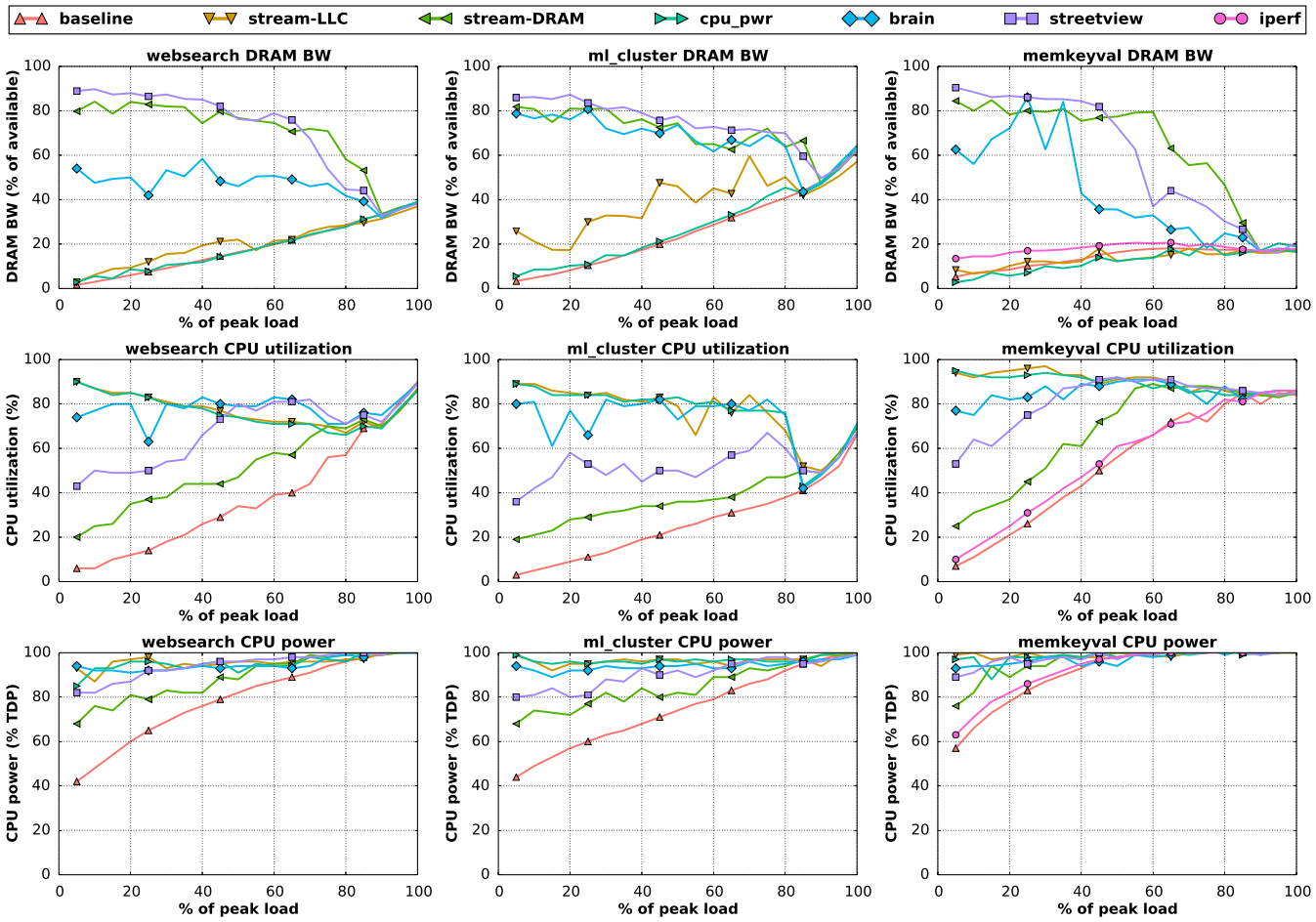

Figure 6. Various system utilization metrics of LC applications co-located with BE jobs under *Heracles*.

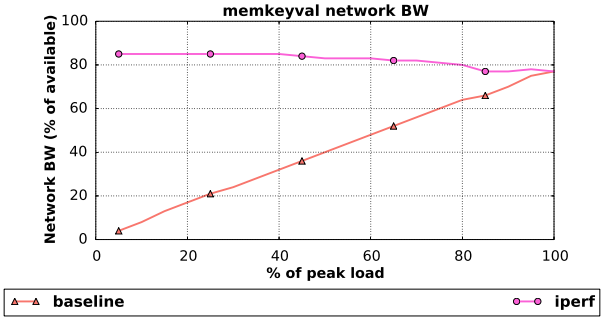

Figure 7. Network bandwidth of *memkeyval* under *Heracles*.

search is not fully loaded and colocation has high potential.

Latency SLO: Figure 8 shows the latency SLO with and without *Heracles* for the 12-hour trace. *Heracles* produces no SLO violations while reducing slack by 20-30%. Meeting the 99%-ile tail latency at each leaf is sufficient to guarantee the global SLO. We believe we can further reduce the slack in larger *websearch* clusters by introducing a centralized controller that dynamically sets the per-leaf tail latency targets based on slack at the root [47]. This will allow a future version of *Heracles* to take advantage of slack in higher layers of the fan-out tree.

Server Utilization: Figure 8 also shows that *Heracles* successfully converts the latency slack in the baseline case into significantly increased EMU. Throughout the trace, *Heracles* colocates sufficient BE tasks to maintain an average EMU of 90% and a minimum of 80% without causing SLO violations. The *websearch* load varies between 20% and 90% in this trace.

TCO: To estimate the impact on total cost of ownership, we use the TCO calculator by Barroso et al. with the parameters from the case-study of a datacenter with low per-server cost [7]. This model assumes $2000 servers with a PUE of 2.0 and a peak power draw of 500W as well as electricity costs of $0.10/kW-hr. For our calculations, we assume a cluster size of 10,000 servers. Assuming pessimistically that a *websearch* cluster is highly utilized throughout the day, with an average load of 75%, *Heracles*' ability to raise utilization to 90% translates to a 15% throughput/TCO improvement over the baseline. This improvement includes the cost of the additional power consumption at higher utilization. Under the same assumptions, a controller that focuses only on improving energy-proportionality for *websearch* would achieve throughput/TCO gains of roughly 3% [47].

If we assume a cluster for LC workloads utilized at an average of 20%, as many industry studies suggest [44, 74], *Heracles* can achieve a 306% increase in throughput/TCO. A controller focusing on energy-proportionality would achieve improvements of

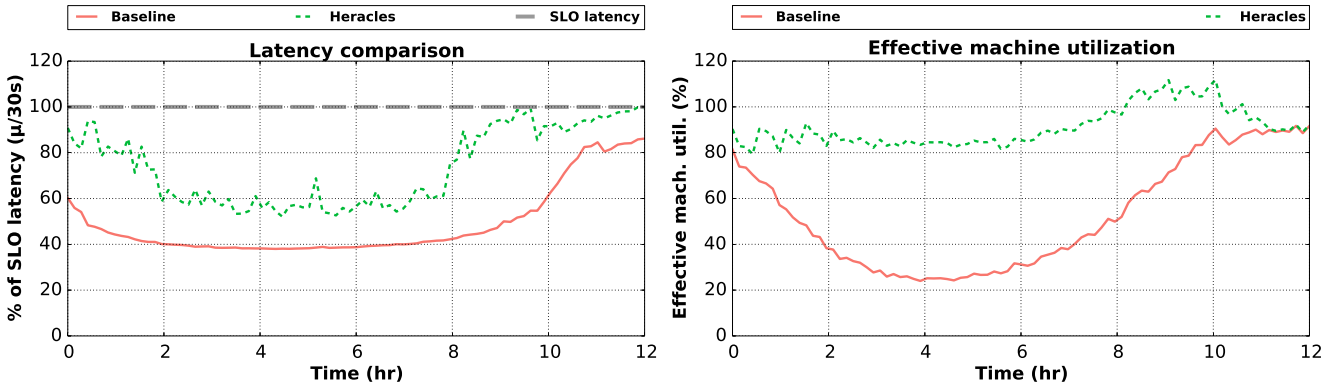

Figure 8. Latency SLO and effective machine utilization for a *websearch* cluster managed by *Heracles*.

less than 7%. *Heracles*' advantage is due to the fact that it can raise utilization from 20% to 90% with a small increase to power consumption, which only represents 9% of the initial TCO. As long as there are useful BE tasks available, one should always choose to improve throughput/TCO by colocating them with LC jobs instead of lowering the power consumption of servers in modern datacenters. Also note that the improvements in throughput/TCO are large enough to offset the cost of reserving a small portion of each server's memory or storage for BE tasks.

6 Related Work

Isolation mechanisms: There is significant work on shared cache isolation, including soft partitioning based on replacement policies [77, 78], way-partitioning [65, 64], and fine-grained partitioning [68, 49, 71]. Tessellation exposes an interface for throughput-based applications to request partitioned resources [45]. Most cache partitioning schemes have been evaluated with a utility-based policy that optimizes for aggregate throughput [64]. *Heracles* manages the coarse-grained, way-partitioning scheme recently added in Intel CPUs, using a search for a right-sized allocation to eliminate latency SLO violations. We expect *Heracles* will work even better with fine-grained partitioning schemes when they are commercially available.

Iyer et al. explores a wide range quality-of-service (QoS) policies for shared cache and memory systems with simulated isolation features [30, 26, 24, 23, 29]. They focus on throughput metrics, such as IPC and MPI, and did not consider latency-critical workloads or other resources such as network traffic. Cook et al. evaluate hardware cache partitioning for throughput based applications and did not consider latency-critical tasks [15]. Wu et al. compare different capacity management schemes for shared caches [77]. The proposed Ubik controller for shared caches with fine-grained partitioning support boosts the allocation for latency-critical workloads during load transition times and requires application level changes to inform the runtime of load changes [36]. *Heracles* does not require any changes to the LC task, instead relying on a steady-state approach for managing cache partitions that changes partition sizes slowly.

There are several proposals for isolation and QoS features for memory controllers [30, 56, 32, 59, 57, 20, 40, 70]. While our work showcases the need for memory isolation for latency-critical workloads, such features are not commercially available at this point. Several network interface controllers implement bandwidth limiters and priority mechanisms in hardware. Unfortunately, these features are not exposed by device drivers. Hence, *Heracles* and related projects in network performance isolation currently use Linux qdisc [33]. Support for network isolation in hardware should strengthen this work.

The LC workloads we evaluated do not use disks or SSDs in order to meet their aggressive latency targets. Nevertheless, disk and SSD isolation is quite similar to network isolation. Thus, the same principles and controls used to mitigate network interference still apply. For disks, we list several available isolation techniques: 1) the cgroups blkio controller [55], 2) native command queuing (NCQ) priorities [27], 3) prioritization in file-system queues, 4) partitioning LC and BE to different disks, 5) replicating LC data across multiple disks that allows selecting the disk/reply that responds first or has lower load [17]. For SSDs: 1) many SSDs support channel partitions, separate queueing, and prioritization at the queue level, 2) SSDs also support suspending operations to allow LC requests to overtake BE requests.

Interference-aware cluster management: Several cluster-management systems detect interference between colocated workloads and generate schedules that avoid problematic colocations. Nathuji et al. develop a feedback-based scheme that tunes resource assignment to mitigate interference for colocated VMs [58]. Bubble-flux is an online scheme that detects memory pressure and finds colocations that avoid interference on latency-sensitive workloads [79, 51]. Bubble-flux has a backup mechanism to enable problematic co-locations via execution modulation, but such a mechanism would have challenges with applications such as *memkeyval*, as the modulation would need to be done in the granularity of microseconds. DeepDive detects and manages interference between co-scheduled applications in a VM system [60]. CPI2 throttles low-priority workloads that interfere with important services [80]. Finally, Paragon and Quasar use online classification to estimate interference and to colocate workloads that are unlikely to cause interference [18, 19].

The primary difference of *Heracles* is the focus on latency-critical workloads and the use of multiple isolation schemes in order to allow aggressive colocation without SLO violations at scale. Many previous approaches use IPC instead of latency as the performance metric [79, 51, 60, 80]. Nevertheless, one can couple *Heracles* with an interference-aware cluster manager in order to optimize the placement of BE tasks.

Latency-critical workloads: There is also significant work in

optimizing various aspects of latency-critical workloads, including energy proportionality [53, 54, 47, 46, 34], networking performance [35, 8], and hardware-acceleration [41, 63, 72]. *Heracles* is largely orthogonal to these projects.

7 Conclusions

We present *Heracles*, a heuristic feedback-based system that manages four isolation mechanisms to enable a latency-critical workload to be colocated with batch jobs without SLO violations. We used an empirical characterization of several sources of interference to guide an important heuristic used in *Heracles*: interference effects are large only when a shared resource is saturated. We evaluated *Heracles* and several latency-critical and batch workloads used in production at Google on real hardware and demonstrated an average utilization of 90% across all evaluated scenarios without any SLO violations for the latency-critical job. Through coordinated management of several isolation mechanisms, *Heracles* enables colocation of tasks that previously would cause SLO violations. Compared to power-saving mechanisms alone, *Heracles* increases overall cost efficiency substantially through increased utilization.

8 Acknowledgements

We sincerely thank Luiz Barroso and Chris Johnson for their help and insight in making our work possible at Google. We also thank Christina Delimitrou, Caroline Suen, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was supported by a Google research grant, the Stanford Experimental Datacenter Lab, and NSF grant CNS-1422088. David Lo was supported by a Google PhD Fellowship.

References

[1] "Iperf - The TCP/UDP Bandwidth Measurement Tool," https://iperf.fr/.

[2] "memcached," http://memcached.org/.

[3] "Intel®64 and IA-32 Architectures Software Developer's Manual," vol. 3B: System Programming Guide, Part 2, Sep 2014.

[4] Mohammad Al-Fares *et al.*, "A Scalable, Commodity Data Center Network Architecture," in *Proc. of the ACM SIGCOMM 2008 Conference on Data Communication*, ser. SIGCOMM '08. New York, NY: ACM, 2008.

[5] Mohammad Alizadeh *et al.*, "Data Center TCP (DCTCP)," in *Proc. of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY: ACM, 2010.

[6] Luiz Barroso *et al.*, "The Case for Energy-Proportional Computing," *Computer*, vol. 40, no. 12, Dec. 2007.

[7] Luiz André Barroso *et al.*, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2nd ed. Morgan & Claypool Publishers, 2013.

[8] Adam Belay *et al.*, "IX: A Protected Dataplane Operating System for High Throughput and Low Latency," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014.

[9] Sergey Blagodurov *et al.*, "A Case for NUMA-aware Contention Management on Multicore Systems," in *Proc. of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. Berkeley, CA: USENIX Association, 2011.

[10] Eric Boutin *et al.*, "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014.

[11] Bob Briscoe, "Flow Rate Fairness: Dismantling a Religion," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 2, Mar. 2007.

[12] Martin A. Brown, "Traffic Control HOWTO," http://linux-ip.net/articles/Traffic-Control-HOWTO/.

[13] Marcus Carvalho *et al.*, "Long-term SLOs for Reclaimed Cloud Computing Resources," in *Proc. of SOCC*, Seattle, WA, Dec. 2014.

[14] McKinsey & Company, "Revolutionizing data center efficiency," *Uptime Institute Symp.*, 2008.

[15] Henry Cook *et al.*, "A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-efficiency While Preserving Responsiveness," in *Proc. of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY: ACM, 2013.

[16] Carlo Curino *et al.*, "Reservation-based Scheduling: If You're Late Don't Blame Us!" in *Proc. of the 5th annual Symposium on Cloud Computing*, 2014.

[17] Jeffrey Dean *et al.*, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, Feb. 2013.

[18] Christina Delimitrou *et al.*, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," in *Proc. of the 18th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, 2013.

[19] Christina Delimitrou *et al.*, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *Proc. of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, UT, 2014.

[20] Eiman Ebrahimi *et al.*, "Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems," in *Proc. of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY: ACM, 2010.

[21] H. Esmaeilzadeh *et al.*, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, June 2011.

[22] Sriram Govindan *et al.*, "Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *Proc. of the 2nd ACM Symposium on Cloud Computing*, 2011.

[23] Fei Guo *et al.*, "From Chaos to QoS: Case Studies in CMP Resource Management," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, Mar. 2007.

[24] Fei Guo *et al.*, "A Framework for Providing Quality of Service in Chip Multi-Processors," in *Proc. of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC: IEEE Computer Society, 2007.

[25] Nikos Hardavellas *et al.*, "Toward Dark Silicon in Servers," *IEEE Micro*, vol. 31, no. 4, 2011.

[26] Lisa R. Hsu *et al.*, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches As a Shared Resource," in *Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '06. New York, NY: ACM, 2006.

[27] Intel, "Serial ATA II Native Command Queuing Overview," http://download.intel.com/support/chipsets/imsm/sb/sata2_ncq_overview.pdf, 2003.

[28] Teerawat Issariyakul *et al.*, *Introduction to Network Simulator NS2*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[29] Ravi Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," in *Proc. of the 18th Annual International Conference on Supercomputing*, ser. ICS '04. New York, NY: ACM, 2004.

[30] Ravi Iyer *et al.*, "QoS Policies and Architecture for Cache/Memory in CMP Platforms," in *Proc. of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '07. New York, NY: ACM, 2007.

[31] Vijay Janapa Reddi *et al.*, "Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, Jun. 2010.

[32] Min Kyu Jeong *et al.*, "A QoS-aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in *Proc. of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY: ACM, 2012.

[33] Vimalkumar Jeyakumar *et al.*, "EyeQ: Practical Network Performance Isolation at the Edge," in *Proc. of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA: USENIX Association, 2013.

[34] Svilen Kanev *et al.*, "Tradeoffs between Power Management and Tail Latency in Warehouse-Scale Applications," in *IISWC*, 2014.

[35] Rishi Kapoor *et al.*, "Chronos: Predictable Low Latency for Data Center Applications," in *Proc. of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. New York, NY: ACM, 2012.

[36] Harshad Kasture *et al.*, "Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads," in *Proc. of the 19th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, March 2014.

[37] Wonyoung Kim *et al.*, "System level analysis of fast, per-core DVFS using on-chip switching regulators," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, Feb 2008.

[38] Quoc Le *et al.*, "Building high-level features using large scale unsupervised learning," in *International Conference in Machine Learning*, 2012.

[39] Jacob Leverich *et al.*, "Reconciling High Server Utilization and Sub-millisecond Quality-of-Service," in *SIGOPS European Conf. on Computer Systems (EuroSys)*, 2014.

[40] Bin Li *et al.*, "CoQoS: Coordinating QoS-aware Shared Resources in NoC-based SoCs," *J. Parallel Distrib. Comput.*, vol. 71, no. 5, May 2011.

[41] Kevin Lim *et al.*, "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached," in *Proc. of the 40th Annual International Symposium on Computer Architecture*, 2013.

[42] Kevin Lim *et al.*, "System-level Implications of Disaggregated Memory," in *Proc. of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, ser. HPCA '12. Washington, DC: IEEE Computer Society, 2012.

[43] Jiang Lin *et al.*, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, Feb 2008.

[44] Huan Liu, "A Measurement Study of Server Utilization in Public Clouds," in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth Intl. Conf. on*, 2011.

[45] Rose Liu *et al.*, "Tessellation: Space-time Partitioning in a Manycore Client OS," in *Proc. of the First USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'09. Berkeley, CA: USENIX Association, 2009.

[46] Yanpei Liu *et al.*, "SleepScale: Runtime Joint Speed Scaling and Sleep States Management for Power Efficient Data Centers," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ: IEEE Press, 2014.

[47] David Lo *et al.*, "Towards Energy Proportionality for Large-scale Latency-critical Workloads," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ: IEEE Press, 2014.

[48] Krishna T. Malladi *et al.*, "Towards Energy-proportional Datacenter Memory with Mobile DRAM," *SIGARCH Comput. Archit. News*, vol. 40, no. 3, Jun. 2012.

[49] R Manikantan *et al.*, "Probabilistic Shared Cache Management (PriSM)," in *Proc. of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC: IEEE Computer Society, 2012.

[50] J. Mars *et al.*, "Increasing Utilization in Modern Warehouse-Scale Computers Using Bubble-Up," *Micro, IEEE*, vol. 32, no. 3, May 2012.

[51] Jason Mars *et al.*, "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations," in *Proc. of the 44th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, ser. MICRO-44 '11, 2011.

[52] Paul Marshall *et al.*, "Improving Utilization of Infrastructure Clouds," in *Proc. of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2011.

[53] David Meisner *et al.*, "PowerNap: Eliminating Server Idle Power," in *Proc. of the 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV, 2009.

[54] David Meisner *et al.*, "Power Management of Online Data-Intensive Services," in *Proc. of the 38th ACM Intl. Symp. on Computer Architecture*, 2011.

[55] Paul Menage, "CGROUPS," https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[56] Sai Prashanth Muralidhara *et al.*, "Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning," in *Proc. of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY: ACM, 2011.

[57] Vijay Nagarajan *et al.*, "ECMon: Exposing Cache Events for Monitoring," in *Proc. of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY: ACM, 2009.

[58] R. Nathuji *et al.*, "Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds," in *Proc. of EuroSys, France*, 2010.

[59] K.J. Nesbit *et al.*, "Fair Queuing Memory Systems," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, Dec 2006.

[60] Dejan Novakovic *et al.*, "DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments," in *Proc. of the USENIX Annual Technical Conference (ATC'13)*, San Jose, CA, 2013.

[61] W. Pattara-Aukom *et al.*, "Starvation prevention and quality of service in wireless LANs," in *Wireless Personal Multimedia Communications, 2002. The 5th International Symposium on*, vol. 3, Oct 2002.

[62] M. Podlesny *et al.*, "Solving the TCP-Incast Problem with Application-Level Scheduling," in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, Aug 2012.

[63] Andrew Putnam *et al.*, "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ: IEEE Press, 2014.

[64] M.K. Qureshi *et al.*, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, Dec 2006.

[65] Parthasarathy Ranganathan *et al.*, "Reconfigurable Caches and Their Application to Media Processing," in *Proc. of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY: ACM, 2000.

[66] Charles Reiss *et al.*, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *ACM Symp. on Cloud Computing (SoCC)*, Oct. 2012.

[67] Chuck Rosenberg, "Improving Photo Search: A Step Across the Semantic Gap," http://googleresearch.blogspot.com/2013/06/improving-photo-search-step-across.html.

[68] Daniel Sanchez *et al.*, "Vantage: Scalable and Efficient Fine-grain Cache Partitioning," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, Jun. 2011.

[69] Yoon Jae Seong *et al.*, "Hydra: A Block-Mapped Parallel Flash Memory Solid-State Disk Architecture," *Computers, IEEE Transactions on*, vol. 59, no. 7, July 2010.

[70] Akbar Sharifi *et al.*, "METE: Meeting End-to-end QoS in Multicores Through System-wide Resource Management," in *Proc. of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '11. New York, NY: ACM, 2011.

[71] Shekhar Srikantaiah *et al.*, "SHARP Control: Controlled Shared Cache Management in Chip Multiprocessors," in *Proc. of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY: ACM, 2009.

[72] Shingo Tanaka *et al.*, "High Performance Hardware-Accelerated Flash Key-Value Store," in *The 2014 Non-volatile Memories Workshop (NVMW)*, 2014.

[73] Lingjia Tang *et al.*, "The impact of memory subsystem resource sharing on datacenter applications," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, June 2011.

[74] Arunchandar Vasan *et al.*, "Worth their watts? - an empirical study of datacenter servers," in *Intl. Symp. on High-Performance Computer Architecture*, 2010.

[75] Nedeljko Vasić *et al.*, "DejaVu: accelerating resource allocation in virtualized environments," in *Proc. of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, 2012.

[76] Christo Wilson *et al.*, "Better Never Than Late: Meeting Deadlines in Datacenter Networks," in *Proc. of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY: ACM, 2011.

[77] Carole-Jean Wu *et al.*, "A Comparison of Capacity Management Schemes for Shared CMP Caches," in *Proc. of the 7th Workshop on Duplicating, Deconstructing, and Debunking*, vol. 15. Citeseer, 2008.

[78] Yuejian Xie *et al.*, "PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches," in *Proc. of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY: ACM, 2009.

[79] Hailong Yang *et al.*, "Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers," in *Proc. of the 40th Annual Intl. Symp. on Computer Architecture*, ser. ISCA '13, 2013.

[80] Xiao Zhang *et al.*, "CPI2: CPU performance isolation for shared compute clusters," in *Proc. of the 8th ACM European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013.

[81] Yunqi Zhang *et al.*, "SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers," in *International Symposium on Microarchitecture (MICRO)*, 2014.